34장. 이터러블 ⚠

34.1 이터레이션 프로토콜

1. ES6에서 도입된 **이터레이션 프로토콜(iteration protocol)**은 순회 가능한(iterable) 데이터 컬렉션
   (자료구조)을 만들기 위해 ECMAScript 사양에 정의하여 미리 약속한 규칙

2. 이터레이션 프로토콜에는
   - 이터러블 프로토콜
   - 이터레이터 프로토콜

34.1.1 이터러블

1. 이터러블 프로토콜을 준수한 객체를 **이터러블**이라 한다.
   즉, 이터러블은 Symbol.iterator를 프로퍼티 키로 사용한 메서드를 직접 구현하거나
   프로토타입 체인을 통해 상속받은 객체를 의미

   [예제 34-01]

```
const isIterable = v => v !== null && typeof v[Symbol.iterator] === 'function

// 배열, 문자열, Map, Set 등은 이터러블이다.
isIterable([]);  // true
isIterable('');  // true
isIterable(new Map());  // true
isIterable(new Set());  // true
isIterable({});  // true
```

- 이터러블인지 확인하는 함수

34.1.2 이터레이터

1. 이터러블의 Symbol.iterator 메서드를 호출하면 이터레이터 프로토콜을 준수한 이터레이터 반환.
   이 이터레이터는 next 메서드를 갖는다.

2. 이터레이터의 next 메서드 : 이터러블의 각 요소를 순회하기 위한 **포인터** 역할.
   즉, next 메서드를 호출하면, 이터러블을 순차적으로 한 단계씩 순회하며 순회 결과를 나타내는
   이터레이터 리절트 객체(iterator result object) 반환

   [예제 34-06]

```
// 배열은 이터러블 프로토콜을 준수한 이터러블이다.
const array = [1, 2, 3];

// Symbol.iterator 메서드는 이터레이터를 반환한다. 이터레이터는 next 메서드 가짐.
const iterator = array[Symbol.iterator]();

// next 메서드를 호출하면 이터러블 순회하며 순회 결과를 나타내는 이터레이터 리절트 객체 반환
// 이터레이터 리절트 객체는 value, done 프로퍼티를 갖는 객체.
console.log(iterator.next());  // { value: 1, done: false }
console.log(iterator.next());  // { value: 2, done: false }
console.log(iterator.next());  // { value: 3, done: false }
console.log(iterator.next());  // { value: undefined, done: true }
```

34.2 빌트인 이터러블

1. 표준 빌트인 객체들은 빌트인 이터러블이다.

빌트인 이터러블	Symbol.iterator 메서드
Array	Array.prototype[Symbol.iterator]
String	String.prototype[Symbol.iterator]
Map	Map.prototype[Symbol.iterator]
Set	Set.prototype[Symbol.iterator]
TypedArray	TypedArray.prototype[Symbol.iterator]
arguments	arguments.prototype[Symbol.iterator]
DOM 컬렉션	NodeList.prototype[Symbol.iterator] HTMLCollection.prototype[Symbol.iterator]

34.3 for ... of 문

1. for ... of 문은 이터러블 순회하면서 이터러블 요소를 변수에 할당한다.

   `for ( 변수선언문 of 이터러블 ) { ... }`

2. for ... of 문은 for ... in 문의 형식과 매우 유사

   `for ( 변수선언문 in 객체 ) { ... }`

3. for ... in 문은 객체 프로토타입 체인 상에 존재하는 모든 프로토타입 프로퍼티 중,

   프로퍼티 어트리뷰트 [[Enumerable]] 값이 true인 프로퍼티를 순회하며 열거한다.

4. [예제 34-07]

```
for (const item of [1, 2, 3]) {
    // item 변수에 순차적으로 1, 2, 3이 할당된다.
    console.log(item);  // 1 2 3
}
```

34.4 이터러블과 유사 배열 객체

1. 유사 배열 객체는 이터러블이 아닌 객체다.

   따라서 유사 배열 객체에는 Symbol.iterator 메서드가 없기때문에 for ... of 문으로 순회할 수 X

2. 단, arguments, NodeList, HTMLCollection은 유사 배열 객체이면서 이터러블.

34.5 이터레이션 프로토콜의 중요성

1. 이터러블은 for ... of 문, 스프레드 문법, 배열 디스트럭처링 할당과 같은 데이터 소비자에 의해 사용되므로 데이터 공급자(data provider)의 역할을 한다고 할 수 있다.

2. 이터레이션 프로토콜은 다양한 데이터 공급자가 하나의 순회 방식을 갖도록 규정하여, 데이터 소비자가 효율적으로 다양한 데이터 공급자를 사용할 수 있도록 **데이터 소비자와 데이터 공급자를 연결하는 인터페이스 역할**을 함.

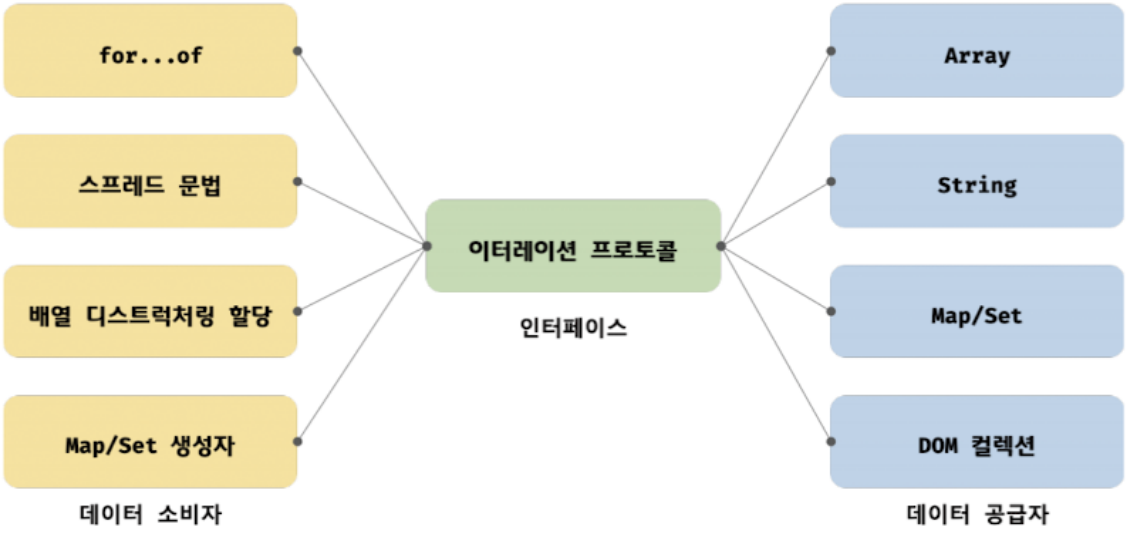

그림 34-2 이터러블은 데이터 소비자와 데이터 공급자를 연결하는 인터페이스

34.6 사용자 정의 이터러블

34.6.1 사용자 정의 이터러블 구현

1. 이터레이션 프로토콜 준수하지 않는 일반 객체도, 이터러블 프로토콜 준수하도록 구현하면

   사용자 정의 이터러블이 된다.

[예제 34-12]

```
// 피보나치 수열을 구현한 사용자 정의 이터러블
const fibonacci = {
  // Symbol.iterator 메서드를 구현하여 이터러블 프로토콜을 준수한다.
  [Symbol.iterator]() {
    let [pre, cur] = [0, 1]; // "36.1. 배열 디스트럭처링 할당" 참고
    const max = 10; // 수열의 최대값

    // Symbol.iterator 메서드는 next 메서드를 소유한 이터레이터를 반환해야 하고
    // next 메서드는 이터레이터 리절트 객체를 반환해야 한다.
    return {
      next() {
        [pre, cur] = [cur, pre + cur]; // "36.1. 배열 디스트럭처링 할당" 참고
        // 이터레이터 리절트 객체를 반환한다.
        return { value: cur, done: cur >= max };
      }
    };
  }
};

// 이터러블인 fibonacci 객체를 순회할 때마다 next 메서드가 호출된다.
for (const num of fibonacci) {
  console.log(num); // 1 2 3 5 8
}

// 직접 순회 방식
// 이터러블의 Symbol.iterator 메서드는 이터레이터를 반환한다.
const iterator = iterable[Symbol.iterator]();

console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: 3, done: false }
console.log(iterator.next()); // { value: 5, done: true }
```

34.6.2 이터러블을 생성하는 함수

1. 전 예제는 최대값 max를 가지고 있다.

   최대값은 고정된 값으로, 외부에서 전달한 값으로 변경할 방법이 없다는 아쉬움 존재

➡️ 수열의 최대값을 인수로 전달받아 이터러블을 반환하는 함수를 만들면 된다.

34.6.3 이터러블이면서 이터레이터인 객체를 생성하는 함수

1. 이터레이터 생성하려면 이터러블의 Symbol.iterator 메서드 호출해야 함.

2. 이터러블이면서 이터레이터인 객체 생성하면 Symbol.iterator 메서드 호출하지 않아도 됨.

```
// 이터러블이면서 이터레이터인 객체
// 이터레이터를 반환하는 Symbol.iterator메서드, 이터레이션 리절트 객체를 반환하는 next 메서
{
    [Symbol.iterator]() { return this; },
    next(){
        return { value: any, done: boolean };
    }
}
```

- Symbol.iterator 메서드와 next 메서드를 소유한 이터러블이면서 이터레이터.

34.6.4 무한 이터러블과 지연 평가

1. 무한 이터러블을 생성하는 함수 → **무한 수열**을 간단히 구현할 수 있다.

```
// 무한 이터러블을 생성하는 함수
const fibonacciFunc = function () {
  let [pre, cur] = [0, 1];

  return {
    [Symbol.iterator]() { return this; },
    next() {
      [pre, cur] = [cur, pre + cur];
      // 무한을 구현해야 하므로 done 프로퍼티를 생략한다.
      return { value: cur };
    }
  };
};

// fibonacciFunc 함수는 무한 이터러블을 생성한다.
for (const num of fibonacciFunc()) {
  if (num > 10000) break;
  console.log(num); // 1 2 3 5 8...4181 6765
}

// 배열 디스트럭처링 할당을 통해 무한 이터러블에서 3개의 요소만 취득한다.
```

```
const [f1, f2, f3] = fibonacciFunc();
console.log(f1, f2, f3); // 1 2 3
```

- 배열, 문자열 등은 모든 데이터를 메모리에 미리 확보한 다음 데이터를 공급, 하지만 위 예제 이터러블은 **지연 평가(Jazy evaluation)**를 통해 데이터 생성.
- **지연 평가** : 데이터가 필요한 시점 이전까지는 미리 데이터 생성 X, 데이터 필요한 시점이 되면 그때야 비로소 데이터를 생성하는 기법.

  즉, 데이터 필요할 때까지 데이터의 생성을 지연하다가, 데이터 필요한 순간 데이터 생성

  ➡️ 이처럼 지연 평가 사용하면 불필요한 데이터 미리 생성하지 않고 필요한 데이터를 필요한 순간에 생성하므로,

  1. 빠른 실행 속도 기대 가능
  2. 불필요한 메모리 소비 X
  3. 무한도 표현할 수 있다

  는 장점이 있다.